UNITED STATES PATENT APPLICATION
FOR
COUNTERING BUFFER OVERRUN SECURITY VULNERABILITIES IN A CPU

INVENTORS:

JAMES W. EDWARDS
a citizen of The United States, residing at
9567 NORTH WEST ARBORVIEW DRIVE PORTLAND, OR 97229

JOHN W. RICHARDSON
a citizen of The United States, residing at
2748 NORTH EAST 19TH AVENUE PORTLAND, OR 97212

YLIAN SAINT-HILAIRE
a citizen of Canada, residing at
1316 NORTH EAST CARLABY WAY #173 HILLSBORO, OR 97124

PREPARED BY:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 WILSHIRE BOULEVARD
SEVENTH FLOOR
LOS ANGELES, CA 90025-1026
(303) 740-1980

EXPRESS MAIL CERTIFICATE OF MAILING

"Express Mail" mailing label number: EL845313368US
Date of Deposit: March 31, 2001
I hereby certify that I am causing this paper or fee to be deposited with the United States Postal Service "Express Mail Post Office to Addressee" service on the date indicated above and that this paper or fee has been addressed to the Commissioner of Patents and Trademarks, Washington, D. C. 20231

April M. Worley
(Typed or printed name of person mailing paper or fee)

(Signature of person mailing paper or fee)

3/31/01
(Date signed)

COUNTERING BUFFER OVERRUN SECURITY VULNERABILITIES IN A CPU

FIELD OF THE INVENTION

[0001] The invention relates generally to the field of computer operating system security. More particularly, the invention relates to preventing security vulnerabilities resulting from buffer overruns.

BACKGROUND OF THE INVENTION

[0002] It is well known that buffer overrun errors are the most common form of errors on the Internet today. Intentional use of buffer overrun errors to attack a system is commonly known as “stack smashing”. Such attacks can cause a system crash, corrupt data, or allow an attacker to execute malicious code on a target machine.

[0003] **Figure 1** is a block diagram illustrating a typical stack. This example depicts a stack 100 after a function has been called. Here, arguments 105, a return address 110, a previous frame pointer 115, and local variables 120 have been placed on the stack 100. The stack 100 also contains a buffer 125 into which data will be placed.

[0004] A stack smashing attack normally occurs by overrunning this buffer 125. A piece of code may contain a string array and allow user input into the array without checking the size of the data entered. For example, an application may contain a fixed string array of 10 characters. Therefore, the buffer will contain space for these 10 characters. If more than 10 characters are written into the array, the buffer 125 will overflow. Once the buffer 125 overflows, the local variables 120, previous frame pointer 115, and return address 110 will be overwritten. Vulnerability occurs when the return address 110 is overwritten. This causes processing to jump back to an unintended location after execution of the called function is finished. In some cases, by placing into the buffer 125 data indicating a specific address, cause processing to return to and execute malicious code that had previously been stored there.

[0005] One of the current solutions involve changing the way a programmer writes code. That is, a programmer should avoid code that allows a user to write data onto the stack without validation. However, this solution only works if programmers consistently write code that does not violate this rule. Another possible solution is to produce compilers that can prevent code from being written in violation of this rule. However, such a solution would likely not be fool proof and compiler settings may be changed to disable such safeguards. Another defense is to limit access to the stack by making portions such as the local variables 120, previous frame pointer 115, and others inaccessible. This solution however limits flexibility since it limits the ability of self-modifying code to place snippets of code onto the stack.

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] The appended claims set forth the features of the invention with particularity. The invention, together with its advantages, may be best understood from the following detailed description taken in conjunction with the accompanying drawings of which:

[0007] **Figure 1** is a block diagram illustrating a typical stack;

[0008] **Figure 2** is a block diagram illustrating an example of a typical computer system upon which embodiments of the present invention may be implemented;

[0009] **Figure 3** is a flowchart illustrating a high-level view of countering buffer overrun security vulnerabilities according to one embodiment of the present invention;

[0010] **Figure 4** is a flowchart illustrating function call processing according to one embodiment of the present invention;

[0011] **Figure 5** is a flowchart illustrating function return processing according to one embodiment of the present invention;

[0012] **Figure 6** is a flowchart illustrating function call processing according to one embodiment of the present invention;

[0013] **Figure 7** is a flowchart illustrating function return processing according to one embodiment of the present invention; and

[0014] **Figure 8** is a flowchart illustrating load or install processing according to one embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

[0015] A method and apparatus are described for preventing security vulnerabilities resulting from buffer overruns. According to one embodiment of the present invention, CALL is modified to place a return address on the stack, and then a random amount of space is added to the stack. This random value is placed in a known place on the stack, or kept in a non-accessible CPU register. The rest of the stack is built normally. When RET is called it finds the number of bytes added to the stack and finds the return address on the stack and returns as normal. This method allows a simple hardware solution that will not be visible to the software, yet provide a powerful deterrent to hackers looking to exploit buffer overrun vulnerabilities in software. Without any software modifications we would be able to deter a significant number of buffer overrun attacks. By affecting components lower on the environment it is possible to influence a larger set of software. For example, it is possible to affect all of the software running on the system without having to change any of the software.

[0016] In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form.

[0017] The present invention includes various methods, which will be described below. The methods of the present invention may be performed by hardware components or may be embodied in machine-executable instructions, which may be used to cause a general-purpose or special-purpose processor or logic circuits programmed with the instructions to perform the methods. Alternatively, the methods may be performed by a combination of hardware and software.

[0018] The present invention may be provided as a computer program product that may include a machine-readable medium having stored thereon instructions that may be used to program a computer (or other electronic devices) to perform a process according to the present invention. The machine-readable medium may include, but is not limited to, floppy diskettes, optical disks, CD-ROMs, and magneto-optical disks, ROMs, RAMs, EPROMs, EEPROMs, magnet or optical cards, flash memory, or other type of media / machine-readable medium suitable for storing electronic instructions. Moreover, the present invention may also be downloaded as a computer program product, wherein the program may be transferred from a remote computer to a requesting computer by way of data signals embodied in a carrier wave or other propagation medium via a communication link (e.g., a modem or network connection).

[0019] **Figure 2** is a block diagram illustrating an example of a typical computer system upon which embodiments of the present invention may be implemented. Computer system 200 comprises a bus or other communication means 201 for communicating information, and a processing means such as processor 202 coupled with bus 201 for processing information. Computer system 200 further comprises a random access memory (RAM) or other dynamic storage device 204 (referred to as main memory), coupled to bus 201 for storing information and instructions to be executed by processor 202. Main memory 204 also may be used for storing temporary variables or other intermediate information during execution of instructions by processor 202. Computer system 200 also comprises a read only memory (ROM) and/or other static storage device 206 coupled to bus 201 for storing static information and instructions for processor 202.

[0020] A data storage device 207 such as a magnetic disk or optical disc and its corresponding drive may also be coupled to computer system 200 for storing information and instructions. Computer system 200 can also be coupled via bus 201 to a display device 221, such as a cathode ray tube (CRT) or Liquid Crystal Display (LCD), for displaying information to an end user. Typically, an alphanumeric input device 222, including alphanumeric and other keys, may be coupled to bus 201 for communicating information and/or command selections to processor 202. Another type of user input device is cursor control 223, such as a mouse, a trackball, or cursor direction keys for communicating direction information and command selections to processor 202 and for controlling cursor movement on display 221.

[0021] A communication device 225 is also coupled to bus 201. The communication device 225 may include a modem, a network interface card, or other well-known interface devices, such as those used for coupling to Ethernet, token ring, or other types of physical attachment for purposes of providing a communication link to support a local or wide area network, for example. In this manner, the computer system 200 may be coupled to a number of clients and/or servers via a conventional network infrastructure, such as a company's Intranet and/or the Internet, for example.

[0022] It is appreciated that a lesser or more equipped computer system than the example described above may be desirable for certain implementations. Therefore, the configuration of computer system 200 will vary from implementation to implementation depending upon numerous factors, such as price constraints, performance requirements, technological improvements, and/or other circumstances.

[0023] It should be noted that, while the steps described herein may be performed under the control of a programmed processor, such as processor 202, in alternative embodiments, the steps may be fully or partially implemented by any programmable or hard-coded logic, such as Field Programmable Gate Arrays (FPGAs), TTL logic, or Application Specific Integrated Circuits (ASICs), for example. Additionally, the method of the present invention may be performed by any combination of programmed general-purpose computer components and/or custom hardware components. Therefore, nothing disclosed herein should be construed as limiting the present invention to a particular embodiment wherein the recited steps are performed by a specific combination of hardware components.

[0024] **Figure 3** is a flowchart illustrating a high-level view of countering buffer overrun security vulnerabilities according to one embodiment of the present invention. Generally, CALL is modified to place a return address on the stack, and then a random amount of space is added to the stack. This random value is placed in a known place on the stack, or kept in a non-accessible CPU register. The rest of the stack is built normally. When RET is called it finds the number of bytes added to the stack and finds the return address on the stack and returns as normal.

[0025] As illustrated in Figure 3, the modified CALL procedure is executed at processing block 305. Details of this processing will be described below with reference to figure 4 and 6. The CALL processing starts the procedure or function called which is then executed at processing block 310. When the called function is finished executing, the modified RET procedure is executed at processing block 315. Details of this processing will be discussed in greater detail below with reference to figures 5 and 7.

[0026] This method makes it significantly more difficult for stack overruns to have an adverse result on the machine. The hope would be that a large percentage of the time the running application would encounter an invalid return pointer in the stack frame causing the application to terminate, rather than allowing the inserted code to run causing a security vulnerability. By crashing the application the system administrator can know that their system is under attack and take appropriate defenses before restarting the application in question and continuing on.

[0027] **Figure 4** is a flowchart illustrating function call processing according to one embodiment of the present invention. First, at processing block 405, the appropriate return address is placed on the stack. Next, at processing block 410, a random number is calculated and the number is saved at processing block 415. This number may be saved on the stack or in a register on the processor that is not generally accessible. At processing block 420, a number of bytes of blank space is placed onto the stack equal to the random number. A stack is then built normally at processing block 425. Finally, at processing block 430, an end of stack pointer is set to the end of the stack frame. This embodiment could be inserted either at compile time by an optimized compiler or even at load time by having the loader find all function entry and exit points and inserting a call to the function that is inserted by the loader. An example of such a process is discussed in greater detail below with reference to figure 8.

[0028] **Figure 5** is a flowchart illustrating function return processing according to one embodiment of the present invention. In this example the random number saved during call processing is recalled at processing block 505. Next, at processing block 510, the number of blank spaces added during call processing are removed from the stack to find the return address. Finally, at processing block 515, the end of stack pointer is set to the end of the previous stack frame.

[0029] The modified call and return processing, combined allow for a simple hardware solution that will not be visible to the software, yet provide a powerful deterrent to hackers looking to exploit buffer overrun vulnerabilities in software. Without any software modifications it is possible to deter a significant number of buffer overrun attacks.

[0030] **Figure 6** is a flowchart illustrating function call processing according to one embodiment of the present invention.This embodiment replaces call/ret sequences with calls to subroutines that are capable of more sophisticated processing, including hashing invariant parts of the stack frame like return address and verifying that the stack frame has not been corrupted.

[0031] As illustrated by figure 6, a return address is placed on the stack at processing block 605. Next, at processing block 610 a hash value of stack frame invariants is calculated. At processing block 615 the hash value is saved in a secure location such as a register on the processor that is not generally accessible. Finally, a stack is built normally at processing block 620.

[0032] This embodiment could be inserted either at compile time by an optimized compiler or even at load time by having the loader find all function entry and exit points and inserting a call to the function that is inserted by the loader. An example of such a process is discussed in greater detail below with reference to figure 8.

[0033] **Figure 7** is a flowchart illustrating function return processing according to one embodiment of the present invention. First, at processing block 705, a hash value of stack frame invariants is calculated. This calculation uses the same hash function and same stack frame invariants as the hash function in call processing. Next, at decision block 710, the call hash value saved during call processing is compared to the return hash value. If the hash values match, the end of stack pointer is set to the end of the previous stack frame at processing block 720. If the hash values do not match, a stack corruption exception is executed at processing block 715.

[0034] **Figure 8** is a flowchart illustrating load or install processing according to one embodiment of the present invention.Here, at either executable load or install time, a search is performed for all function calls at processing block 805. At processing block 810 a random amount of space is added to the stack frame at each function call. All references to the stack are then adjusted at processing block 815 to compensate for the added space. If this process is performed during executable installation, the executable is then saved to disk at optional processing block 820.

[0035] Various software and hardware embodiments will have very different performance issues. For example, calling a function on every function entry and exit will add significant processor overhead, but also provide a huge guarantee that the stack frame is completely intact. Making modifications at load time will allow a different version of the application to run each time it is executed at the cost of having to add and compensate for blank space added to the stack every time the executable is loaded, leading to significantly longer load times. Tradeoffs can be made between time and reliability depending on specific system requirements.